

# Implementation and Performance

---

## Compile-Time

`.java -> .class` compiler

Compiles down to standard "bytecode" format in a `.class` file

class file

Encodes all sorts of info: class names, method names, ivar names and types  
Much more info than in C/C++. You can use the "Reflection" language features to access this info at run-time from your code.

bytecode

optimized to be: compact, portable/unambiguous  
describes a computation -- like `.pdf` describes an image  
not necessarily optimized for speed

## Run-Time

Class Loader

Maps class names like "TetrisPiece" to pull the right thing out of the file system.

You need to mess with this, for example, to have a "plug-ins" folder that you load classes from.

Verifier

Checks that the bytecode is correct -- e.g. a method that claims to return something really does have a return of a pointer at its end

Run-Time Checks

Types, array bounds, etc. are checked during the run of the program -- the verifier represents the structure checks that can be done just once -- everything else is checked at run-time.

Security Manager

Provides fine-grain control over which classes can access which resources -- like a mini-operating system. Your code can plug in a custom security manager.

## Big Picture

Robust/Safe

Portable

## "Programmer Efficiency" Fast?

### Robust Niche

#### "90% Robust" Doesn't Work

There are no short-cuts to Java's airtight robustness

#### There is only one "100%" robust niche

Java makes a lot of compromises to be 100% robust and portable.

Makes the language very unique

### Tradeoff

#### Ambitious Language Design

Design an ambitious language for the state-of-the-art hardware.

#### 90% -> 100% upgrade NO!

It's hard to retrofit or evolve robustness over time -- you have to design it in right from the start.

When the language starts out, it should push the current hardware to the limit.

e.g. C++

It's not practical to "evolve" C++ to be as robust or flexible as Java.

### Evolve or Die

#### Design does not evolve (Inertia)

(As above) the fundamental constraints of say, the .class format (Java) or .o (C/C++), are hard to evolve.

Software inertia makes change hard.

#### Implementation Speed -- improve over time

With the fundamentally right design, you can evolve and improve the implementation for better performance over time -- just write better and better JIT technology over time.

### Steve Jobs Idea

Software develops slower than its hardware

MacOS vs PowerPC

DOS/Windows vs. x86

End up with old, legacy software holding back quickly evolving hardware

Therefore: make the software very ambitious at the start compared to the current state of hardware  
 Your software decisions will introduce lots of inertia -- be ambitious enough that the software system will be great even a few years out.

## ●●●Performance

### JITs

Just In Time compiler  
 Translate the bytecode into native code -- do a hasty, low quality job of it.  
 Big improvement, but uses lots of memory

### HotSpot

Watch the code, and figure out what to optimized.  
 Compile a small amount of the code aggressively.  
 Do lots of inlining -- enables all sorts of other optimizations

## Optimization 101

### Reality

Hard to predict where the bottlenecks are  
 It's not so hard to use tools to measure what the code is doing once it is written.  
 Therefore, write the code the way you want to be **correct** and **finished** first, then worry about optimization.

### "Premature Optimization" = evil

Classic advice from Don Knuth  
 Write the code to be straightforward and correct first  
 Maybe it's fast enough already  
 If not, measure to find the bottleneck  
 Focus optimization there. Use CS161 type optimal algorithms + use language techniques as below

### Data Structures

Your data structure will have a profound influence on performance.  
 This is one bit of "early" design where you might want to think about performance a little.  
 The choice of data structure (what you store, who has pointers to whom) can be very constraining on the possible algorithms later. on.

### Proportionality To Caller

Suppose we write a foo() utility in a way which is easy to code but naive -- it currently costs 1 millisecond, but could be sped up drastically. foo() is only called in one place by the bar() method.  
 How do you know if this matters?

The key question: **how costly is bar()**? If bar() takes 20 milliseconds, then foo() just doesn't matter. The smart strategy is to leave foo() in it's slow/naive/correct implementation -- find something else to fix.  
If bar() takes 2 milliseconds, then foo() makes a huge difference and should be fixed.

## 1-1 User Event Rule

If something happens some fixed number of times like 1 or 3, for each single user event, such as a button push, then performance is not too important for that operation.

Watch for operations that happen 100's or thousands of times in relation to each user event.

User events happen very slowly from the computer's point of view.  
e.g. We didn't worry about paint() too much for Tetris, but we did worry about place/undo (many times for each piece as the brain plays).

## Algorithm Optimization

### Pixels Expensive

Laying down pixels is costly

It's worth having an algorithm that is smart enough to only draw what's necessary

### Disk Expensive

Getting bytes of the disk or network is expensive

### Computing Again and Again

Sometimes all the fancy abstraction and encapsulation can create an algorithm that's pretty stupid: compute foo(x) add it to y. Compute foo(x) again add it to z...

You can use encapsulation here where the client is unaware that the second call to getFoo() is just returning a cached answer.

## Java Tips

Using the right algorithm is the most important. After that we have language feature rules...

### 1. 1-10-1000 Rule

assignment (=) : 1 unit of time

method call : 10 units of time

similar overhead to C

new object or array : 1000 units of time

Newer VMs are making this cheaper, but it's still much more expensive than other operations

## 1. int getWidth() vs. Dimension getSize()

getSize() requires a heap allocated object

getWidth() and getHeight() may just be inlined to move the two ints right into the local vars of the caller code.

With HotSpot, supposedly short lived objects have been implemented to much faster, so this may be less important in the future.

## 2. static buffer -- "singleton"

Suppose you need some temporary array in a method.

Instead of calling `new char[1000]` in every call...

1. allocate a static array just once, and use it every time
2. (better) declare a static array, and allocate it the first time the method is called by checking if it's null -- avoids creating more load-time cost

Note: be careful if the method is executed by multiple threads

## 3. clever swapping

108 Tetris board implementation

Allocate two copies of the "board" data structure.

Swap between the two implement the undo feature

Point: rotate between a fixed number of objects, to avoid ever needing to call new

### Cache

In this case, the use of cache memory is better as well -- the two copies get "hot" and we just switch between them.

## 4. Locals Faster Than IVars

Local variables are faster than member variables of any object (the receiver or some other object). Locals are also easier for the optimizer to work with for a variety of optimizations.

This could be a `.width` variable in some other object, or in this receiver -- they are both slower than a local stack variable.

Inside loops, pull needed values into local variables (`int i`);

Suppose we are in a for loop...

1. Slow -- message send

`...i < piece.getWidth()`

2. Medium -- instance variable -- with a JIT, this case and (1) above are essentially the same.

`...i < piece.width`

-or-

`...i < width` (suppose the code is executing against the receiver)

3. Fast -- pull the state into a local (stack) variable, and then use it. This allows the implementation to pull the value into a native register. If the value is in an ivar, the runtime needs to retrieve it from memory every time it is used. It's hard for the runtime to deduce that `.width` is not being changed, so it has to reload it from memory. Whereas it's easy for it to deduce that `localWidth` is not being changed, so it can just put it in a

register and use that value the whole time. (Note theme for the future: we're sensitive to generating memory traffic.)

```
int localWidth = piece.getWidth(); // or width if we are the reciever
... i<localWidth...
```

-or-

```
// make it even more clear for the JIT...
final int localWidth = piece.getWidth();
```

## 5. Avoid Synchronized

Synchronized has a moderate runtime cost -- although this has been reduced as of Java 2

Can have synch and unsynch versions of the same method, and switch between the two based on some other flag.

Use "immutable" (unchangeable) objects to finesse synchronization problems.

## 6. StringBuffer

Use StringBuffer to put together strings -- change to String only once it's not going to change.

### Automatic

This case the compiler optimizes for you -- appending together a bunch of strings at one moment into one immutable string.

```
String s = "a string" + foo.toString() + "some other string";
```

### No

```
String record; // ivar
```

```
void transaction(String id) {
    record = record + " " + id; // NO, chews through memory
}
```

### YES

```
StringBuffer record;
void transaction(String id) {
    record.append(" ");
    record.append(id); + id;
}
```

## 7 Don't Parse

Slow: read in XML, ASCII, etc. -- build big data structure

Fast: read it into memory, but leave it as just chars. Do the search, etc. in the chars -- just parse/build what you need on the fly.

## 8. Avoid Weird Code

The whole stack of VM optimizations added over time will be oriented towards common looking code -- write your code in the most obvious, common way, not some weird way

e.g. `for (int i = 0; i < bound; i++) {...}`

Also, realize that obvious method implementations like `getWidth()` `{return(width);}` will certainly be targeted by HotSpot, so don't worry about the method overhead.

## 9. Don't Use Vector

It's too synchronized, use the new Collections ArrayList instead.

If you can get away with a plain array, even better -- that's the fastest

## 10. Use final for Methods

VM optimizers, and hot spot in particular, make aggressive use of inlining called code into its caller code.

Inlining enables many other optimizations.

Pro: "final" is a huge aid in enabling inlining

Con: subclasses can no longer override your method. If you're just compiling all of your own code together, then it's no great loss.

## Naive...



